1. **Define Agility. Explain its principals.**

Agility, in the context of software development, refers to the ability to respond rapidly and adaptively to change. The principles of agility are outlined in the passage as follows:

* Customer Satisfaction: The highest priority is to satisfy the customer through early and continuous delivery of valuable software.
* Embracing Change: Welcome changing requirements, even late in development, to harness change for the customer's advantage.
* Frequent Delivery: Deliver working software frequently, with a preference for shorter timescales.
* Collaboration: Business people and developers must work together daily throughout the project.
* Motivated Individuals: Build projects around motivated individuals and provide them with the necessary support.
* Effective Communication: Face-to-face conversation is the most efficient method of conveying information within a development team.
* Measuring Progress: Working software is the primary measure of progress.
* Sustainable Development: Agile processes promote sustainable development to maintain a constant pace indefinitely.
* Technical Excellence: Continuous attention to technical excellence and good design enhances agility.
* Simplicity: Maximizing the amount of work not done is essential for agility.
* Self-Organizing Teams: The best architectures, requirements, and designs emerge from self-organizing teams.
* Reflection and Improvement: Regularly reflect on how to become more effective and adjust behavior accordingly.

1. **Explain requirements engineering steps in brief.**

Requirements Engineering involves several stages, each aimed at ensuring a clear understanding of what needs to be developed:

1. \*\*Inception\*\*:

- Establish a basic understanding of the problem, including:

- Who needs a solution?

- What kind of solution is desired?

- How effectively have the customer and developer communicated so far?

2. \*\*Elicitation\*\*:

- Gather requirements from all stakeholders involved.

3. \*\*Elaboration\*\*:

- Create an analysis model that breaks down requirements into data, function, and behavioral aspects.

4. \*\*Negotiation\*\*:

- Reach an agreement on a system that is feasible for both developers and customers.

5. \*\*Specification\*\*:

- Document the requirements in one or more formats:

- Graphical models

- Formal mathematical models

- User scenarios (use-cases)

6. \*\*Validation\*\*:

- Review the requirements to check for:

- Errors or misinterpretations

- Areas needing clarification

- Missing information

- Inconsistencies or conflicts, especially in larger projects

7. \*\*Requirements Management\*\*:

- Implement activities to identify, control, and track requirements and changes throughout the project.

- Use tools like features traceability tables, source traceability tables, dependency traceability tables, subsystem traceability tables, and interface traceability tables to manage requirements effectively.

1. **Explain the elicitation step in requirements engineering**

Elicitation is a crucial phase in requirements engineering, serving as the foundation for successful software development. This step involves **gathering, discovering, and understanding the needs** and expectations of **stakeholders** to define the system's requirements accurately. Various techniques are employed to extract valuable information, ensuring that the final requirements align with the stakeholders' objectives. Here are some common **elicitation techniques**:

1. Interviews:

- Description: Conducting one-on-one or group interviews with stakeholders to obtain insights into their **needs, expectations, and concerns.**

- Application: Direct interaction allows for a detailed exploration of stakeholder perspectives, providing rich qualitative data.

2. Surveys and Questionnaires:

- Description: Distributing structured surveys to a larger group of stakeholders to collect information **systematically**.

- Application: Useful for gathering a **broad range of opinions efficiently**, especially when dealing with a large number of stakeholders.

3. Workshops:

- Description: Organizing collaborative workshops or **brainstorming sessions** to encourage active participation and generate innovative ideas.

- Application: Fosters open communication and allows stakeholders to collectively contribute to requirement identification and refinement.

4. Observation:

- Description: Actively observing users in their natural environment to understand their behavior, tasks, and challenges.

- Application: Provides **real-world insights** into user interactions and uncovers implicit requirements that might not be explicitly stated.

5. Prototyping:

- Description: Developing prototypes or mockups to provide stakeholders with a **tangible representation of the system.**

- Application: Helps in **visualizing requirements** and allows stakeholders to provide **feedback on the system's design and functionality.**

6. Document Analysis:

- Description: Reviewing existing documentation, such as **business plans and reports,** to extract relevant information about requirements.

- Application: Aids in understanding the **organization's context, objectives, and constraints.**

7. Focus Groups:

- Description: Gathering a **diverse group of stakeholders** for open discussions and feedback sessions on requirements.

- Application: Encourages the expression of **various perspectives**, leading to a more comprehensive understanding of stakeholder needs.

8. Use Cases and Scenarios:

- Description: Developing use cases or scenarios to explore how users will interact with the system and identifying specific requirements in different situations.

- Application: Provides detailed insights into system behavior and user interactions, helping to define functional requirements.

9. Ethnographic Studies:

- Description: Conducting in-depth studies of the **organization's culture** and practices to gain insights into the system's operational context.

- Application: Helps in understanding the socio-cultural factors that may impact system requirements.

In summary, the elicitation step involves a combination of these techniques to ensure a holistic and accurate understanding of stakeholder needs. Each technique contributes unique perspectives, enriching the overall requirements gathering process. The goal is to establish a solid foundation for the subsequent stages of requirements engineering, ultimately leading to the successful development of a system that aligns with stakeholder expectations.

1. **Explain in detail the Intermediate COCOMO model**

**OR 16. Explain (any) COCOMO I model**

COCOMO (Constructive Cost Model) is a widely used software cost estimation model developed by Barry Boehm. COCOMO I is the initial version of this model, and it helps project managers and software developers estimate the effort, time, and cost required for a software project.

**1. Basic Idea:**

- COCOMO I is based on the idea that the effort and cost required for a software project are directly related to the size and complexity of the project.

**2. Three Modes:**

- COCOMO I has three modes, each suited for different types of software projects:

- **Organic Mode:** Ideal for small, well-understood projects with experienced developers. The emphasis is on flexibility and creativity.

- **Semi-Detached Mode:** Suitable for medium-sized projects with a mix of experienced and less-experienced developers.

- **Embedded Mode:** Appropriate for large, complex projects where strict requirements and tight controls are necessary.

EXPLAIN INTERMEDIATE COCOMO Model in detail

**Intermediate COCOMO Model:**

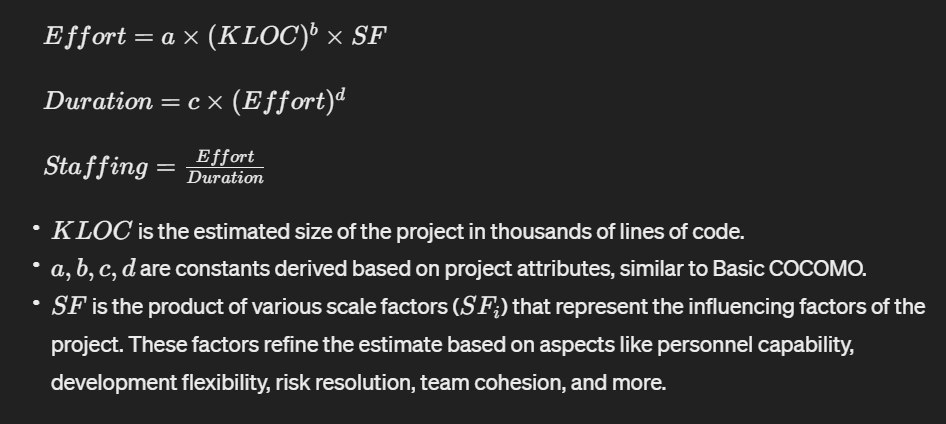
**Overview:**

- Intermediate COCOMO, also known as COCOMO 1.5, bridges the gap between the simplicity of Basic COCOMO and the complexity of Advanced COCOMO.

- It is designed for projects that fall between the characteristics of small, well-understood projects (suited for Basic COCOMO) and large, highly complex projects (suited for Advanced COCOMO).

**Equation:**

The effort estimation equation in Intermediate COCOMO is an extension of the Basic COCOMO equation, considering additional factors or scale factors.



**Scale Factors :**

Intermediate COCOMO introduces several scale factors, each representing a specific aspect of the project that can impact the effort required. These factors include:

**1. Precedentedness:** Reflects the familiarity of the project with similar ones.

**2. Development Flexibility:** Measures the degree of flexibility in the development process.

**3. Risk Resolution:** Evaluates the effectiveness of risk management practices.

**4. Team Cohesion:** Reflects the degree of collaboration and unity within the development team.

**5. Process Maturity**: Considers the maturity level of the development process.

**Example:**

Suppose you are estimating a project with a size of 50,000 lines of code (\(KLOC = 50\)). You determine the constants \(a, b, c, d\) based on historical data for your type of project. Additionally, you assess the scale factors (\(SF\_i\)) based on the project's characteristics, such as the team's experience, development flexibility, and risk resolution. Plug these values into the Intermediate COCOMO equation to estimate the effort, duration, and staffing.

**Advantages:**

- Intermediate COCOMO provides a more refined estimation by considering additional project characteristics.

- It allows for flexibility in adapting the estimation process to the specific attributes of the project.

**Limitations:**

- While more detailed than Basic COCOMO, Intermediate COCOMO may not capture the full complexity of certain projects.

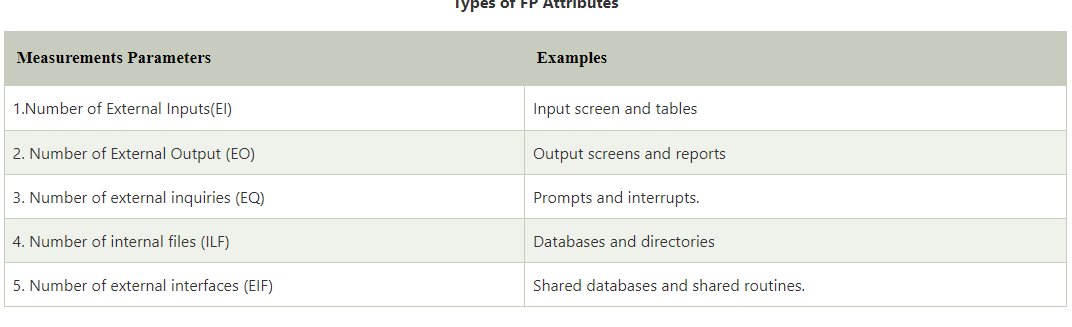
- The accuracy of the estimate still relies on the accuracy of the input parameters and historical data.

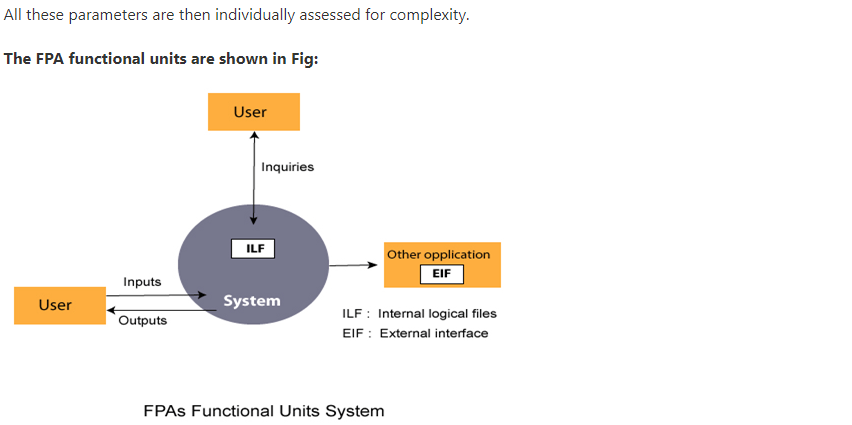
In summary, Intermediate COCOMO enhances the estimation process by incorporating scale factors to address the nuances of project characteristics, providing a more nuanced and detailed approach compared to Basic COCOMO.

**10. Explain FP estimation method with suitable diagram**

Function points are a **unit of measurement** used to quantify the functionality provided by a software application. They are **independent of the programming language and technology used**. Function points are calculated based on the **user inputs, outputs, queries, files, and interfaces in a system.**

The process of determining function points involves **assigning weights** to different components of the software, such as inputs, outputs, inquiries, files, and interfaces. These weights are then multiplied by the number of occurrences of each component, and the results are summed to obtain the total function points.





External inputs — password, panic button, and activate/deactivate

External inquiries — zone inquiry and sensor inquiry.

Internal Logical Files - system configuration file.

External outputs - messages and sensor status

External Interface Files - test sensor, zone setting, activate/deactivate, and alarm alert

**Q - Compute the function point, productivity, documentation, cost per function for the following data:**

**Number of user inputs = 24**

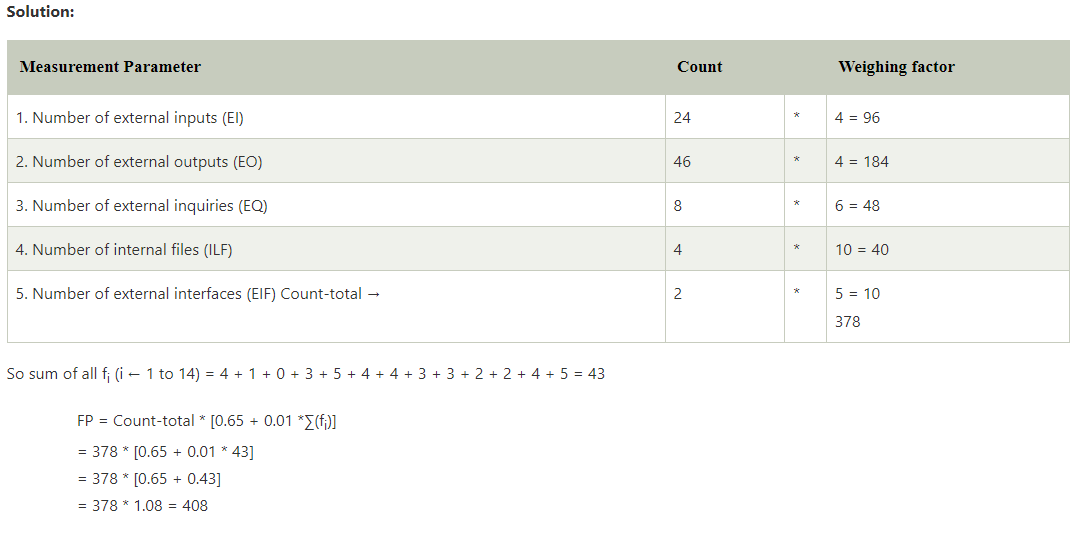
**Number of user outputs = 46**

**Number of inquiries = 8**

**Number of files = 4**

**Number of external interfaces = 2**

**Various processing complexity factors are: 4, 1, 0, 3, 3, 5, 4, 4, 3, 3, 2, 2, 4, 5.**



1. **Explain in brief the different types of coupling and cohesion. Give one practical example of high cohesion and low coupling**

**Coupling:**

**Definition (1 mark):**

Coupling is a measure of how interconnected modules are in a software structure. It reflects the extent to which modules rely on each other.

Importance (1 mark):

In software design, the goal is to minimize coupling. Low coupling simplifies understanding, reduces the ripple effect of errors, and enhances system stability.

**Types of Coupling (2.5 marks):**

**1. No Direct Coupling (0.5 marks):**

- Modules are unrelated and subordinate to different modules, showing no direct connections.

- For instance, Modules 'a' and 'd' are independent and have no direct coupling.

**2. Data Coupling (0.5 marks):**

- Modules are connected through a simple argument list, passing minimal data.

- In the example, Module 'c' is subordinate to Module 'a' with a straightforward argument list.

**3. Stamp Coupling (0.5 marks):**

- A variation of data coupling, where a part of a data structure is passed between modules.

- Illustrated in the connection between Modules 'b' and 'a' in the provided content.

**4. External Coupling (0.5 marks)**:

- Modules are coupled to an external environment, like I/O coupling to specific devices and protocols.

- Necessary but should be limited to maintain simplicity.

**5. Common Coupling (0.5 marks):**

- Multiple modules reference a global data area.

- Modules 'c,' 'g,' and 'k' accessing a shared data item in the content demonstrate common coupling.

**6. Content Coupling (0.5 marks):**

- The highest degree of coupling, occurring when one module uses data or control information within another.

- Should be avoided due to its potential complexity and difficulties in diagnosis.

**Cohesion:**

**Definition (1 mark):**

Cohesion is the degree to which elements within a module work together to achieve a single function. High cohesion is desirable for a well-designed system.

**Importance (1 mark):**

Cohesion ensures that a module focuses on a specific task, enhancing maintainability and reducing error propagation.

**Types of Cohesion (2.5 marks):**

**1. Functional Cohesion (0.5 marks):**

- Elements of a module cooperate to achieve a single function.

- For example, a module handling user authentication performs a single function.

**2. Sequential Cohesion (0.5 marks):**

- Elements form a sequence, with output from one being input to the next.

- Seen in modules where steps are performed in a specific order.

**3. Communicational Cohesion (0.5 marks):**

- All tasks refer to or update the same data structure.

- Modules operating on a shared array or stack demonstrate communicational cohesion.

**4. Procedural Cohesion (0.5 marks):**

- Elements are parts of a procedure with a specific sequence for achieving a goal.

- Illustrated with the algorithm for decoding a message.

**5. Temporal Cohesion (0.5 marks):**

- Functions are associated by the fact that all must be executed at the same time.

- In the content, activities related in the same time (BOOT SETUP) showcase temporal cohesion.

**6. Logical Cohesion (0.5 marks):**

- All elements of the module perform a similar operation.

- Modules for error handling, data input, and data output exemplify logical cohesion.

**7. Coincidental Cohesion (0.5 marks):**

- Tasks are associated loosely, if at all.

- In the content, the error processing module performing loosely related tasks is coincidentally cohesive.

**Practical Examples (1 mark):**

- High Cohesion and Low Coupling Example:

- Consider a module responsible for handling user authentication in a web application.

- High Cohesion: The module performs a single function - verifying user credentials.

- Low Coupling: It interacts minimally with other modules, only accessing a database for user information.

In essence, effective software design strives for high cohesion within modules and low coupling between them, promoting simplicity, maintainability, and stability.

* **Explain the various design concepts in software engineering.**
* **Discuss Abstraction, Information Hiding and Functional Independence.**

**1. Abstraction (2 marks):**

- Abstraction in software design involves hiding unnecessary details to reduce complexity and improve efficiency.

- It works at different levels – a broad description at a higher level and a detailed one at a lower level of abstraction.

- For example, when using a TV remote, you press a button without needing to understand the complex electronics inside.

**2. Modularity (1.5 marks):**

- Modularity is about dividing a software system into smaller, independent parts to reduce complexity.

- This helps in creating, modifying, and reusing components independently, enhancing maintainability.

- An analogy is a car engine – it has various components working independently but contributing to the overall function.

**3. Architecture (1.5 marks**):

- Architecture in software design is the technique of designing the structure of the system, focusing on the interaction of its components.

- Think of it like planning the layout of rooms and corridors in a building to ensure a smooth flow. In software, it's about organizing data and elements.

**4. Refinement (1 mark**):

- Refinement is the process of presenting software in detail, removing impurities, and improving quality.

- It's akin to revising an essay to correct errors and enhance clarity without changing the main message.

**5. Pattern (0.5 marks):**

- In software design, a pattern is a repeated form or design solving common recurring problems.

- Similar to using a common strategy in a game for different levels, patterns provide consistency and efficiency in software solutions.

**6. Information Hiding (0.5 marks):**

- Information hiding involves designing modules to hide data from unauthorized access.

- Like having a password-protected folder, it ensures that sensitive information is accessible only to those with proper credentials.

**7. Refactoring (0.5 marks):**

- Refactoring is reconstructing software without changing its behavior to reduce complexity.

- An analogy is rearranging a room's furniture for better organization without altering the room's primary function.

**Effective Modular Design (3 marks):**

* **Discuss on Modularity and Functional Independence fundamentals of design concepts.**

**- Role (1 mark**):

- Effective modular design is crucial in software engineering to manage complex systems.

- It involves breaking down a software system into modules, making it easier to develop, modify, and maintain.

- **Modularization (1 mark):**

- Modularization is the process of creating multiple independent modules.

- These modules should be solvable, modifiable, and compilable separately for effective modular design.

**- Functional Independence (1 mark)**:

- Functional independence means each function in a module performs a single task with minimal interaction with other modules.

- This independence contributes to good design, affecting the overall quality of the software.

**Benefits of Independent Modules (1 mark):**

**- Clear Requirements (0.5 marks):**

- Breaking down functionality into atomic levels provides clear requirements for developers.

- This simplifies the design process and reduces errors.

**- Limited Dependency (0.5 marks):**

- Independent modules have limited or no dependency on others, allowing changes without affecting the entire system.

**- Measurement (0.5 marks):**

- Independence is measured using Cohesion and Coupling criteria.

- High cohesion and low coupling indicate a well-designed, modular software system.